CVGIP: GRAPHICAL MODELS AND IMAGE PROCESSING
Vol. 54, No. 3, May, pp. 252–258, 1992

NOTE

Methods for Fast Morphological Image Transforms Using Bitmapped Binary Images

REIN VAN DEN BOOMGAARD AND RICHARD VAN BALEN

Department of Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Received September 24, 1990; accepted December 17, 1991

In this paper we present new implementations for morphological binary image processing on a general-purpose computer, using a bitmap representation of binary images instead of representing binary images as bitplanes inserted in gray value images. The bitmap data representation is a very efficient one, both in terms of memory requirements and in terms of algorithmic efficiency because of the CPU operates on 32 pixels in parallel. The algorithms described in this paper are capable of performing the basic morphological image transforms using structuring elements of arbitrary size and shape. In order to speed up morphological operations with respect to commonly used, large, convex structuring elements, the *logarithmic decomposition* of structuring elements is used. Experiments indicate that the new algorithms are more than 30 times faster for pixelwise operations and about an order of magnitude faster for the basic morphological transforms than the fastest known software implementations. © 1992 Academic Press, Inc.

1. INTRODUCTION

Mathematical morphology as introduced by Serra [1] is currently an important theoretical basis for low-level image processing. This is reflected in the fact that mathematical morphological image transforms can be found in most software packages for image processing. Most morphological transforms are constructed from elementary morphological operations such as erosion, dilation, and the hit-or-miss transform. The speedup of the elementary operations is therefore important in many applications.

In this paper we propose a new algorithmic implementation of the basic binary morphological operations on general-purpose sequential computers that is between 10 and 50 times faster than the fastest existing implementations [2, 3]. In contrast with the previous fast implementations, the proposed method admits the use of structuring elements of arbitrary size and shape. This is based on our belief that implementations of the basic morphological transforms should be valid for arbitrarily shaped structuring elements without strong performance degradation, as noncircular structuring elements prove to be very useful in solving specific problems.

Instead of representing binary images as bitplanes inserted in gray value images, we use the bitmap representation. This representation is very efficient both in terms of memory requirements and in terms of algorithmic efficiency because the CPU operates on 32 pixels in parallel. Besides the use of a bitmap representation to speed up the elementary morphological operations, we also use the logarithmic decomposition of large convex structuring elements, in contrast with existing fast implementations, which use the linear decomposition.

Experimental results presented in this paper indicate that the combined use of a bitmap representation and the logarithmic decomposition of structuring elements leads to very fast algorithms for the basic morphological image transforms.

2. ELEMENTS FROM MATHEMATICAL MORPHOLOGY

We assume that the reader is acquainted with the basic principles of mathematical morphology. For a short introduction the reader is referred to recently published tutorials [4, 5]. In this section we specify the notation used in this paper and discuss some properties that are needed in this paper. Table 1 summarizes the notations for the basic set (i.e., binary image) operations.

In the next section we discuss only the implementation of the hit-or-miss transform, of which both the erosion and dilation are special cases. From the definition of the hit-or-miss transform it can be easily seen that $X \ominus \check{S} = X \otimes (S, \varnothing)$. Given the duality of the erosion and dilation ($X \oplus S = (X^c \ominus S)^c$), the dilation can be written as $X \oplus \check{S} = (X \otimes (\varnothing, S))^c$.

In morphological image processing there is often a need for erosions and dilations using large convex structuring elements. It is a well-known fact in mathematical morphology that convex sets are divisable with respect to the dilation; i.e., if S is convex then $S \oplus S \oplus \cdots \oplus S$ (n

TABLE 1
Notations and Definitions of the Basic Morphological Set Operations

Name	Notation	Definition
Translation	X_t	$X_t = \{x \mid x - t \in X\}$
Complement	X^c	$X^c = \{x \mid x \notin X\}$
Transpose	$\check{X}$	$\check{X} = \{x \mid -x \in X\}$
Union	$X \cup Y$	$X \cup Y = \{x \mid x \in X \text{ or } x \in Y\}$
Intersection	$X \cap Y$	$X \cap Y = \{x \mid x \in X \text{ and } x \in Y\}$
Difference	X/Y	$X/Y = X \cap Y^c$
Dilation	$X \oplus \check{S}$	$X \oplus \check{S} = \bigcup_{y \in \check{S}} X_y$
Erosion	$X \ominus \check{S}$	$X \ominus \check{S} = \bigcap_{y \in \check{S}} X_y$
Hit-or-miss transform	$X \otimes (S, T)$	$X \otimes (S, T) = (X \ominus \check{S}) \cap (X^c \ominus \check{T})$

Note. We use X, Y, S, and T to denote sets in n-dimensional space and we use x, y, and t to denote a position vector in that space.

terms) has the same shape as S but is n times larger. In this paper where we deal only with discrete sets, we therefore define

$$nS = \underbrace{S \oplus S \oplus \cdots \oplus S}_{n \text{ terms}}. \tag{1}$$

Let $S \subset \mathbf{Z}^2$ and let $\#(S)$ denote the number of pixels in the set S; then $\#(nS) \approx n^2\#(S)$. The time complexity (denoted t) of most algorithms for erosions and dilations is linear dependent on the number of pixels in the structuring element used. Thus for an erosion or dilation using nS we have $t = \mathbb{O}\,(n^2\#(S))$. Using the associativity of the dilation, $X \oplus (S \oplus T) = (X \oplus S) \oplus T$, the dilation with respect to $n\check{S}$ can be written as

$$X \oplus nS = (\cdots((X \underbrace{\oplus S) \oplus S)\cdots) \oplus S}_{n \text{ terms}},$$

showing that $X \oplus nS$ can be calculated by repeating the dilation using structuring element S, n times. The time complexity to calculate $X \oplus nS$ using the above *linear decomposition* of nS is $t_{\text{lin}} = \mathbb{O}\,(n\#(S))$.

Although linear decomposition as well as logarithmic decomposition (introduced below) can be used for all convex sets, we restrict ourselves in this paper to *symmetric* sets. Decomposition of nonsymmetric n-dimensional convex sets will be treated in a separate paper [6]. A set S is said to be symmetric iff $\exists t : \check{S} = S_t$. It has been shown by Pecht [7] that for a two-dimensional convex symmetric set S, the linear decomposition of nS (given in Eq. (1)) can be replaced with

$$nS = S \oplus \underbrace{E(S) \oplus \cdots \oplus E(S)}_{(n-1) \text{ terms}}, \tag{2}$$

where $E(S)$ is the extreme set of S. For discrete sets the extreme set $E(S)$ contains just the vertices of the—continuous—convex hull of S. Using the decomposition in Eq. (2) the time complexity to calculate $X \oplus nS$ is $t_{\text{linextr}} = \mathbb{O}\,(\#(S) + (n-1)\#(E(S)))$. This leads to more efficient algorithms since in general $\#(E(S)) < \#(S)$.

The linear decomposition using extreme sets can be replaced with the more efficient *logarithmic decomposition*. For a symmetric convex set, we can use the linear decomposition using the extreme set $2S = S \oplus E(S)$. Because $\forall m : mS$ is also convex and symmetric we can decompose $4S$ as $4S = 2S \oplus E(2S)$. Substituting $S \oplus E(S)$ for $2S$ we obtain $4S = S \oplus E(S) \oplus E(2S)$. Again $4S$ is convex and symmetric and thus $8S = 4S \oplus E(4S) = S \oplus E(S) \oplus E(2S) \oplus E(4S)$. In general for $n = 2^m$ we have

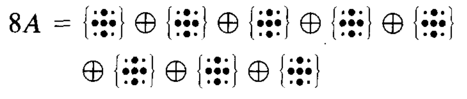
$$2^mS = S \oplus E(S) \oplus E(2S) \oplus E(4S) \oplus \cdots \oplus E(2^{m-1}S).$$

To decompose nS for arbitrary n we can use the above decomposition using $m = \lfloor \log_2(n) \rfloor$ followed by a dilation with respect to structuring element $E((n - 2^m)S)$:

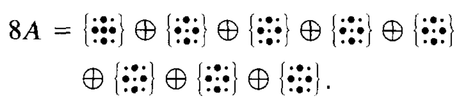
$$nS = S \oplus E(S) \oplus E(2S) \oplus E(4S) \oplus \cdots \oplus E(2^{m-1}S) \oplus E((n - 2^m)S). \tag{3}$$

For a discrete set S, $E(mS)$ contains the same number of pixels as $E(S)$, and thus the time complexity to implement the above decomposition is $t_{\log} = \mathbb{O}\,(\#(S) + (\lfloor \log_2(n) \rfloor + k)\#(E(S)))$, where $k = 0$ iff $n = \lfloor \log_2(n) \rfloor$ and $k = 1$ iff $n \neq \lfloor \log_2(n) \rfloor$.

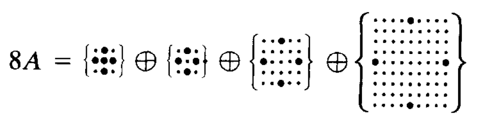
To illustrate the three different approaches to decompose a convex and symmetric set nS, consider the decomposition of $8A$, where A is the diamond-shaped set (or 4-connected neighborhood). The linear decomposition (Eq. (1)) of $8A$ is given by



with a total number of pixels in the structuring elements of 40. The linear decomposition of $8A$ using the extreme set (Eq. (2)) gives (total number of pixels is 33)



Finally the logarithmic decomposition (Eq. (3)) of $8A$ results in



with a total of 17 pixels in the structuring elements.

Logarithmic decomposition is a data-independent method to speed up morphological image processing when large convex structuring elements are used, because it does not depend on the image to be processed. In a later section we compare our algorithms with the data-dependent, *contour-processing* algorithms proposed by Verwer and van Vliet [2]. We introduce the contour-processing algorithms via the propagation. The contour-processing algorithm is used in our algorithm for the propagation. The propagation of a *seed* set X_0 using structuring element S under the *mask* set Y is defined as

$$X_{i+1} = (X_i \oplus S) \cap Y. \tag{4}$$

The propagation with respect to the structuring elements $S = Q$ (the square-shaped or 8-connected neighborhood) or $S = A$ (the diamond-shaped or 4-connected neighborhood) is used to fill the interior or exterior of binary objects. (For a bounded set Y, i.e., of finite spatial extent, we have that for $i \to \infty$: $X_{i+1} = X_i$.) For $S = A$ or $S = Q$ the propagation (Eq. (4)) can be rewritten as:

$$\begin{aligned} R_{i+1} &= (Y \cap (R_i \oplus S))/X_i \\ X_{i+1} &= X_i \cup R_{i+1}, \end{aligned} \tag{5}$$

where $R_0 = X_0/(X \ominus S)$, the contour of X_0, serves as the seed of the propagation. In the above algorithm we dilate R_i instead of X_i. Maintaining a list of all the pixel coordinates of the elements of R_i makes it possible to dilate only those pixels that need dilating. Because R_i in general contains far fewer elements than X_i, this results in faster algorithms. Note that while dilating R_i we can build the list of all pixels in R_{i+1} on the fly. If we take the mask set Y to be the entire image plane ($Y = \mathbf{Z}^2$), propagation equals dilation. Therefore, to calculate $X \oplus nS$ with $S = A$ or $S = Q$ we can use Eq. (5); i.e., $X_n = X \oplus nS$. The time complexity in this case is linear proportional to the number of pixels in the contour R_i (for $i = 0, n-1$), which makes the contour-processing algorithm data dependent.

3. ALGORITHMIC IMPLEMENTATION

3.1. *Data Representation*

In most modern general-purpose computers the efficiency of low-level image-processing operations is bound by the speed at which the image data can be transported between the memory and the CPU. Traditionally in image processing binary images are stored as one bitplane in an $N \times M$ pixel-mapped gray value image. To access the value of one bit in a bitplane the CPU must fetch all the bits of one gray value pixel (typically 8 or 16) and discard all irrelevant bits.

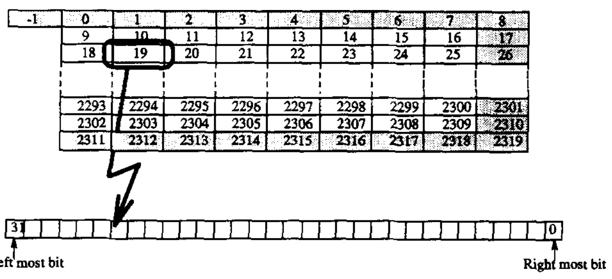


FIG. 1. Bitmap image representation. Only a small part of a binary image of 256 × 256 pixels is shown. The shaded words are not part of the actual image but rather define a border around the image. The word with index 19 is magnified to show the 32 pixels. For this image we have $N = M = 256$, LINEOFFSET = 9, and MAXINDEX = 2319.

In this paper, a bit-mapped representation of binary images is advocated. Each addressable word contains the bit values of 32 horizontally consecutive pixels. A 32-bit computer is optimally used in the sense that all bits fetched in one memory cycle are relevant. Furthermore the algorithms described in this paper operate on 32 pixels in parallel, leading to fast algorithms.

All the words are stored in a linear array in the computer memory. To facilitate the development of efficient algorithms we choose to store the pixels in the image itself, as well as a border around the image. Figure 1 shows the linear array (depicted in a two-dimensional layout). The numbers in the words are the indices in the linear array. The border words depicted at the right of the image also serve as 32-pixel border at the *left* of the actual image (word 17 is the right neighbor of word 16 *and* the left neighbor of word 18). The word with index -1 is used to store the northwest neighboring pixel of the leftmost pixel in word 9.

Let B denote the linear array of bitwords corresponding with an $N \times M$ image. The pixels in the image are denoted with cartesian coordinates (k, l) with $(0, 0)$ being the top-left pixel. In the bitmap we can denote each pixel with its *bitmap coordinates* (i, b), where i is the index in the bitmap array and b is the bitnumber for that particular pixel. Given (k, l) it is then easy to calculate the corresponding (i, b). If we define LINEOFFSET $= 1 + \lceil N/32 \rceil$ and MAXINDEX $=$ LINEOFFSET $\cdot (M + 2) - 1$ then (using "mod" to denote the modulo operation)

$$\begin{aligned} i &= \text{LINEOFFSET} \cdot (l + 1) + \left\lfloor \frac{k}{32} \right\rfloor \\ b &= 31 - (k \bmod 32). \end{aligned} \tag{6}$$

Any image representation must deal with the fact that structuring elements near the edge only partly overlap with the image. In our algorithms, pixels not within the actual image are taken to be either 1 or 0 depending on

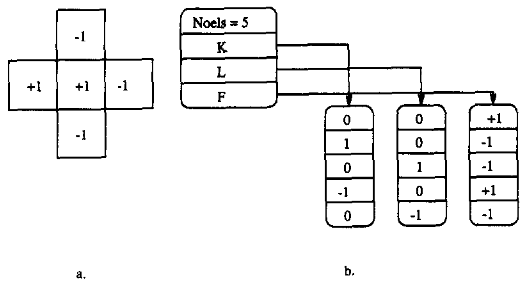


FIG. 2. Hit-or-miss mask representation. (a) An example of a simple hit-or-mask (S, T) with pixels in S denoted as $+1$ and those in T as -1. (b) The corresponding mask representation.

the operation. Note however that the border around the actual image, which is also stored in memory, makes *border checking* unnecessary for structuring element pixels with coordinates (k, l) satisfying $|k| \le 32$ and $|l| \le 1$. In the algorithms described in this paper we assume that $|k| \le 32$, necessitating border checks only for those pixels in a structuring element for which $|l| > 1$. In case $|k| > 32$ extra border checks are necessary. These checks are implemented, but are not shown in the algorithms presented in this paper. The asymmetry caused by storing 32 border pixels at the left and right boundary of the image but only 1 pixel at the top and bottom boundary is introduced because boundary checking in the vertical direction can be performed much faster than boundary checking in the horizontal direction.

Although structuring elements are sets and can be represented with bitmaps, we choose an alternative representation that is more efficient for the algorithms described in the next subsection. Figure 2 depicts the data structure used to represent arbitrary hit-or-miss masks (S, T). The ith element of a hit-or-miss mask is described with the coordinates $(K[i], L[i])$ and its *indicator* $F[i]$. By definition a pixel with index i is an element of S iff $F[i] > 0$ and an element of T iff $F[i] < 0$.

3.2. *Implementation of the Pixelwise Logical Operations*

The pixelwise logical operations of one or two binary input images are the boolean equivalents of the union, intersection, and complement of sets. Algorithm 1 shows the pseudocode for performing the logical AND of two

```
for i=0 to MAXINDEX do
    D[i] = B[i] & C[i]
enddo
```

ALGORITHM 1. Pixelwise logical operations. The pseudocode for performing the logical AND of two images (represented in bitmaps B and C) resulting in bitmap D.

binary images (we use the C syntax to denote AND with "&"). Replacing & with "|" (denoting the bitwise OR) in Algorithm 1 gives the set union. Note that not only the image itself is processed, but also the border. This small amount of extra work makes nested for-loops unnecessary.

3.3. *Implementation of the Morphological Operations*

Switching from the set to the boolean representation of binary images, let p_i $(i = 1, \ldots, n)$ denote the boolean values of the pixels in S_x, and let q_i $(i = 1, \ldots, m)$ denote the boolean values of all pixels in T_x. If we denote the pixel value at position x in $X \otimes (S, T)$ by r we have

$$r = p_1 \,\&\, p_2 \,\&\, \cdots \,\&\, p_n \,\&\, \tilde{q}_1 \,\&\, \tilde{q}_2 \,\&\, \cdots \,\&\, \tilde{q}_m,$$

where $\tilde{q}$ denotes the boolean negation of q.

Let B be the bitmap array representing the input image X and let C be the bitmap representing the image $X \otimes (S, T)$. The mask (S, T) is encoded with the arrays K, L, and F as described in the previous subsection. If j is an index into the mask data structure, then if $0 \le i + L[i] \cdot \text{LINEOFFSET} \le \text{MAXINDEX}$, the word $B[i + L[j] \cdot \text{LINEOFFSET}]$ contains the 32 pixels $L[j]$ lines above (if $L[j] < 0$) or below (if $L[j] > 0$) the 32 pixels in $B[i]$. Consider the hit-or-miss transform $X \otimes (S, T)$,

$$X \otimes \begin{Bmatrix} -1 \\ \underline{+1} \\ -1 \end{Bmatrix},$$

where the central pixel (the origin of the structuring element) is underlined; then for any word $C[i]$ in the resultant bitmap we can write

$$C[i] = B[i] \,\&\, \tilde{B}[i - \text{LINEOFFSET}] \,\& \\ \tilde{B}[i + \text{LINEOFFSET}].$$

In case we are interested in the 32 pixels not exactly above or below the pixels in $B[i]$ (i.e., $0 < K[j] \le 32$) we must combine the bits from two words in the bitmap array. To do so we define the BARRELSHIFT function. This function takes as parameters the bitmap array (B), the index (i) of the word containing the 32 central pixels, and the coordinates $((K[j], L[j]))$ relative to the central pixels of the word that we are interested in. The BARRELSHIFT function is given in Algorithm 2, where we use "<<" and ">>" to denote the shifting of bits in an integer word. The pseudocode for the hit-or-miss transform using arbitrary structuring elements (with the restriction that $\forall j$: $|K[j]| \le 32$) is given in Algorithm 3.

Both the erosion and the dilation are special cases of the hit-or-miss transform (as explained in Section 2). Be-

```
BARRELSHIFT( A, i, k, l, border )
do
    $j = i + l * $ LINEOFFSET
    if ( $0 < j <$ MAXINDEX ) do
        // The pixels are within the bitmap
        if ( $k = 0$ )
            return ( $A[j]$ )
        else if ( $k < 0$ )
            // we are looking for pixels to the left
            // of the central pixel
            return ( $(A[j] >> abs(k))|(A[j-1] << (32+k))$ )
        else
            // we are looking for pixels to the right
            // of the central pixel
            return ( $(A[j] << k)|(A[j+1] >> (32-k))$ )
    else
        // the pixels are outside the bitmap
        return ( border )
    endif
  enddo
enddo
```

ALGORITHM 2. Barrelshift function. Pseudocode for the Barrelshift function, where A is the bitmap, i is the bitmap index of the central pixels, (k, l) is the pixel position relative to these central pixels, and border is the word returned when pixels outside the image are accessed.

cause they are so frequently used, special code has been written with the unnecessary parts in the hit-or-miss transform algorithm omitted.

4. EVALUATION

In this section the performance of the proposed algorithms is compared with the performance of existing implementations. The new algorithms are referred to as the BITMAP algorithms. They are compared with two software implementations and with a special-purpose Image-Processing System (KONTRON, IPS, Eching b, Munich, West Germany) that is referred to as SIP. All our BITMAP algorithms are coded in C. Tests were performed on a SUN SparcStation 1, except for the results obtained with the SIP. All the timing results are obtained by averaging the results over at least 20 experiments. Because some implementations of the morphological transforms are data dependent, we have chosen a set of three test images. These are depicted in Fig. 3.

```
HIT_OR_MISS( A, B, K, L, F )
do
    $i =$ LINEOFFSET // index the first word in the actual image
    for $y = 0$ to $M - 1$ do
        for $x = 0$ to $\lfloor n/32 \rfloor$ do
            $result =$ BITWORDALL // all bits set to 1
            for $j = 0$ to NOELS - 1 do
                if $F[j] > 0$ do
                    $result = result$ & BARRELSHIFT$(A, i, K[j], L[j], 1)$
                else
                    $result = result$ & BARRELSHIFT$(A, i, K[j], L[j], 0)$
                endif
            enddo
            $B[i] = result$
            $i = i + 1$ // index next word
        enddo
        $i = i + 1$ // skip over the word in the border
    enddo
enddo
```

ALGORITHM 3. Hit-or-miss transform. Pseudocode for the hit-or-miss transform, where A and B are the bitmaps representing, respectively, the input image and the resultant image; K, L, F, are the arrays representing the hit-or-miss mask.

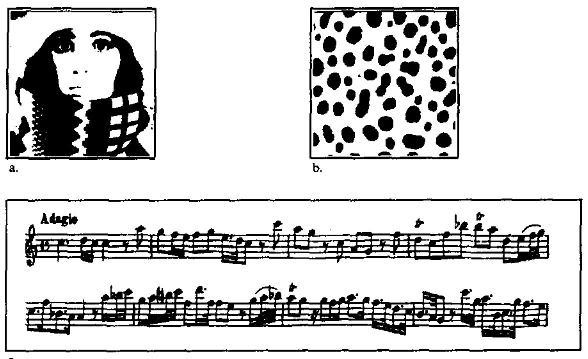


FIG. 3. Test images: (a) 256 × 256 binary image ("trui"), (b) 256 × 256 binary image ("cermet"), and (c) 1024 × 244 binary image ("musicscore").

In the timing results presented in this section we have not included the time that it takes to convert a bitplane in a pixelmap image into a bitmap image and vice versa. This is based on the observation that in most image-processing applications once a gray value image is converted to a binary image (using an appropriate segmentation technique), the conversion back to a gray value image is done only (if at all) after many binary image operations are performed.

4.1. *Pixelwise Logical Operations*

The pixelwise logical operations on bitmap images are much faster than the equivalents working on pixel-mapped images. This is due to the fact that instead of operating on one pixel value at a time, we operate on 32 pixels in parallel. Table 2 shows the experimental results for a 256 × 256 binary image. The algorithm for the pixel mapped images is denoted PIXELMAP. The table also shows the results for the SIP machine.

TABLE 2
Timing Results for the Pixelwise Logical Operations

Operation	BITMAP (ms)	PIXELMAP (ms)	SIP (ms)
AND	2.0	87.3	93.0
OR	1.7	87.3	93.0
EXOR	1.7	115.0	91.0
INVERT	1.3	74.0	NA
COPY	1.3	74.3	46.0

Note. All timings are obtained using a 256 × 256 image. Timing results for the INVERT operation on the SIP were not available.

4.2. *Morphological Transforms*

The bitmap implementations of the erosion are compared with two existing software implementations and with the special-purpose image-processing machine (SIP). The simplest method to implement erosions and dilations using structuring elements of size 3 × 3 is to use look-up tables. A table with 512 entries holds the results (for erosion, dilation, etc.) for all possible configurations in a 3 × 3 neighborhood. Transforming the image then boils down to one table-look action per image pixel. This method is denoted the Straight Forward Implementation (SFI).

We also compare our algorithms with the contour-processing algorithms developed by Verwer and van Vliet [2] (see Section 2), referred to as the VVV algorithms. The first iteration of erosions and dilations using the VVV algorithm is slow because the list containing the pixels to be processed needs to be built by scanning the entire image. Subsequent iterations are much faster (proportional with the number of contour pixels).

Table 3 shows the experimental results for the 4-connected dilation ($X \oplus nA$) for several values of n. For the BITMAP algorithms, only the results using the logarithmic decomposition of nA are shown. In Fig. 4a the timing results for the 4-connected dilation (using the image in Fig. 3a are shown. For the BITMAP algorithm the results using both the linear decomposition (labeled BITMAP (lin)) and the logarithmic decomposition (labeled BITMAP (log)) are shown. The data dependency of the VVV algorithms can be observed in this figure: for $n \approx 32$ the set $X \oplus nA$ fills the entire image plane, at which point the VVV algorithms ends as the list with pixels to be dilated is empty.

In Fig. 4b the timings for the bitmap algorithms are shown again, this time using a linear time scale. Also shown is the function $t(n) = \beta + \alpha \log_2(n)$ with α and β calculated with a least-squares fit using only the experimental timing results for $n = 2, 4, 8, 16, 32$. The deviation of the experimental timings for $n > 32$ is due to the fact that the bitmap data structure is optimized for structuring elements of maximal 65 pixels wide. For larger sizes test must be made to ensure correct behavior at image borders. Note that "vertical" border checking is necessary

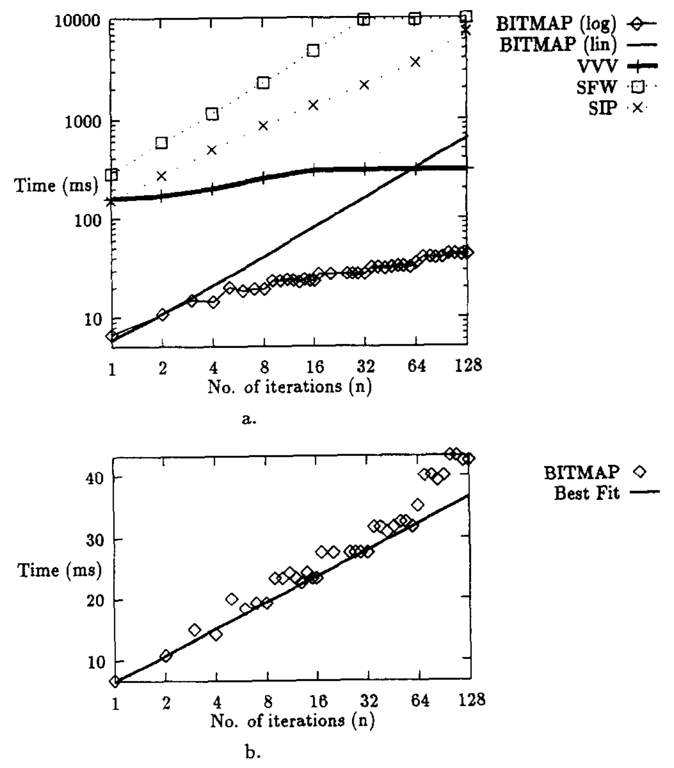


FIG. 4. Timing results for the 4-connected dilation. (a) The results for all implementations described in Table 3. All timing were obtained using the image in Fig. 3a. (b) Only the results for the Bitmap algorithms are repeated, this time using a linear time scale. Also the best-fit line through the points $n = 2, 4, 8, 16$, and 32 is shown. The deviation of the experimental timing from the best fit line for $n > 64$ is due to the fact that the bitmap data structure is optimized for structuring elements that fit into a 65 × 65 window.

TABLE 3
Timing Results for the 4-Connected Dilation

	BITMAP (ms)		VVV (ms)			SFI (ms)			SIP (ms)
Number of iterations	a,b	c	a	b	c	a	b	c	a,b
1	6	23	175	160	639	283	283	1,216	153
2	11	43	187	171	723	583	516	2,416	273
4	15	62	206	200	851	1150	1133	4,900	491
8	19	82	237	255	945	2300	2366	9,850	858
16	24	100	269	302	1089	4716	4800	20,066	1371
32	28	121	288	305	1303	9600	7000	41,616	2155
64	36	153	291	305	1414	9600	6983	86,016	3643
128	44	187	291	305	1423	9750	7050	151,066	7178

Note. Shown are the execution times (in ms) for the 4-connected dilation for the three different original images (denoted a,b and c; see Fig. 3).

TABLE 4
Timing Results for the 4-Connected Propagation

Number of iterations	BITMAP (ms)	VVV (ms)
2	46	133
∞	23	322

Note. Shown are the execution times (in ms) for the 4-connected propagation $X_{i+1} = (X_i \oplus S) \cap Y$. As seed X_0 the skeleton of image "trui" is used. The mask Y is the image "trui" itself. An infinite number of iterations means that the propagation is iterated until $X_{i+1} = X_i$.

for $n \geq 4$ but from Fig. 4 we can conclude that it does not decrease the performance as much as "horizontal" border checking does.

In Table 4 the results for the 4-connected propagation are shown. The bitmap propagation is compared only with the VVV algorithm for propagation. In practice the propagation is used with either very few iterations or with an "infinite" number of iterations (i.e., until nothing changes). We developed two algorithms for the propagation; the first is a direct implementation of Eq. (4). The second uses the contour-processing method (see Eq. (5)) and is used only for an infinite number of iterations.

5. DISCUSSION AND CONCLUSIONS

In this paper we presented new algorithms for binary morphological transforms based on a bitmap representation of binary images and the use of logarithmic decomposition of structuring elements. The new algorithms prove to be on average one order of magnitude faster than existing fast implementations on general-purpose, sequential computers.

We strongly favor the point of view of Piper and Rutovitz [8] that any image-processing software should choose the data representation best suited for processing. The results presented in this paper indicate that bitmap representation of binary images is well suited for morphological image processing, as it is very efficient both in terms of memory requirements and in terms of algorithmic efficiency. Rather than allocating eight or more binary images (when the pixel mapped representation is used) only the bits in one binary image are stored. When working with very large images like those acquired with page scanners, the reduced memory requirement is an important factor.

For dealing with large convex structuring elements (obtained from autodilation of smaller structuring elements) logarithmic decomposition proves to be a powerful tool.

ACKNOWLEDGMENTS

We thank Arnold Smeulders for the valuable suggestions that he made during the preparation of this paper and Ton ten Kate for the timing results of the Kontron Special Purpose Image Processing System. We thank Frans Groen, who always believed that image processing using general-purpose computers could be done very fast, for his encouragement.

REFERENCES

1. J. Serra, *Image Analysis and Mathematical Morphology*, Vol. 1, Academic Press, London, 1982.
2. B. J. H. Verwer and L. J. van Vliet, A contour processing method for fast binary neighbourhood operations, *Pattern Recognition Lett.*, **7**, 1988, 27–36.
3. F. C. A. Groen and N. J. Foster, A fast algorithm for cellular logic operations on sequential machines. *Pattern Recognition Lett.* **2**, 1984, 333–338.
4. R. M. Haralick, S. R. Sternberg, and X. Zhuang, Image analysis using mathematical morphology, *IEEE Trans. Pattern Recognit. Mach. Intelligence* **PAMI-9**, July 1987, 532–550.
5. P. Maragos, Tutorial on advances in morphological image processing and analysis, *Opt. Engrg.* **26**(7), 1987, 623–632.
6. Rein van den Boomgaard, Logarithmic decomposition of convex sets and it use in mathematical morphology, submitted for publication.
7. J. Pecht, Speeding up succesive Minkowski operations, *Pattern Recognition Lett.* **3**(2), 1985, 113–117.
8. J. Piper and D. Rutovich, Data structures for image processing in a C-language and Unix environment, *Pattern Recognition Lett.* **3**(2), 1985, 119–129.